EVMS: A Common Framework for Volume Management

Steven Pratt
Linux Technology Center
IBM
Austin, TX 78758
slpratt@us.ibm.com
http://evms.sf.net

Abstract

The Enterprise Volume Management System (EVMS) brings a new model of volume management to Linux. EVMS integrates all aspects of volume management into a single cohesive package. By introducing a new pluggable architecture, EVMS provides extendibility while ensuring consistency and cooperation across multiple volume management schemes.

EVMS consists of two main components, the Runtime, which resides in the kernel and handles discovery and I/O functions, and the Engine, which resides in User Space and handles setup and configuration. Packaged with the Engine are three user interfaces, a GTK based GUI, a command line interpreter, and an ncurses based interface.

EVMS borrows from the existing Linux volume management technologies, combining them into a single easy to use package. Imagine being able partition your disk, create mirrors and raid devices, define volume groups and logical volumes, all from one integrated, easy to use interface. With EVMS you can do this and more.

EVMS provides immediate benefit to system administrators who wish to get a handle on their storage configurations, as well as less technical users who have not memorized all of the various commands and config files. No more scanning through raidtab files or issuing multiple LVM commands just to find out how a system is configured. Just bring up the EVMS GUI and have all of this information at your fingertips. Not only can you see what disks or partitions make up which volumes, but you can also see if these volumes are formatted or mounted.

1 Introduction

Volume management is an integral part of any operating system. Every major server operating system has some form of volume management capability. The methods vary from simple DOS partitioning to complex volume groups and everything in between. What stands out about each operating system is that they generally have a single method for performing volume management and with that a single consistent interface for configuring storage. When we look at Linux we see that, as with most components, it offers multiple choices for volume management. In most cases having competing products that give the user different capabilities and functions is good for Linux as it fosters competition and technical advancement. In the cases where the user chooses only one of the competing technologies there is really no downside to this approach. However, in the case where similar or competing technologies do not completely overlap, users may desire to use multiple technologies in order to get a combination of function not found in any one technology. In these cases some problems can arise.

This is the case with volume management in Linux today. The three major volume management schemes available in Linux today (partitions, LVM and MD) each provide exclusive features which are not available in the other volume managers.

This means that the user, instead of choosing one method of volume management, may instead use many. This puts the burden on the user/administrator to learn how to use each set of tools. Not only that, but he must figure out any interdependencies that exist and work around them.

To further complicate the task of managing storage, it is not enough to define the storage layout. Usually you must also associate a file system with the volume. This adds another layer in which multiple choices are available, each with slightly different features and interfaces. Today there is little to no coordination between volume managers and the file systems for operations such as resize, where issuing the commands in the wrong order can cause data loss.

EVMS attempts to make sense out of this jumble of components by providing a consistent architecture and framework in which all of the varying technologies can exist in harmony with each other. By providing a consistent set of APIs and common services, EVMS allows new technologies to be added to the existing framework while ensuring that they will interact correctly with all of the current functionality.

2 EVMS Architecture

In EVMS the task of volume management has been divided into three main components, the Engine, the Runtime, and the User Interfaces. EVMS uses a layered, plug-in model in the Engine and Runtime to provide flexibility and extensibility for managing storage. This method of managing storage allows for easy expansion or customization of various levels of volume management. The EVMS framework is fully 64 bit enabled and architecture independent. It is also endian neutral, except for certain compatibility plug-ins for which the original implementation used native layouts.

Runtime

The EVMS Runtime refers to the in-kernel portion of the system. The Runtime has two primary purposes:

1. Coordinating the discovery of logical volumes and creating the necessary block devices to represent those volumes in user-space.

2. Handling I/O to the volumes.

The EVMS kernel component consists of two parts, the common services and the plug-ins. The common services provide the framework which makes up the heart of the EVMS architecture. This framework coordinates the loading and registration of plug-ins for each of the four plug-in classes supported and the in-kernel discovery process. The common services also provide IOCTL routing and helper routines for the plug-ins to use. Before looking at the common services we should first describe each of the classes of plug-ins and how they work.

EVMS kernel plug-ins are built as standard Linux kernel modules and are compiled as part of the Linux kernel build process. These plug-ins add functionality to the EVMS runtime by providing support for specific features of volume management. The plug-ins register with the EVMS common services at init or module load time.

The first class of plug-in is the Device Managers. Device managers are responsible for determining the available devices on the system and presenting them to the EVMS framework. Device managers are also responsible for determining attributes of the devices (such as hardsector size) as well as detecting if a device has been removed or modified. This is the only class of plug-in which deals directly with device drivers or device driver queues. There is currently only one Device Manager implemented (Local Device Manager) which manages all devices found on the gendisk list. It remains to be seen if other device managers are required for features such as multipath IO, NAS, and SAN, or whether

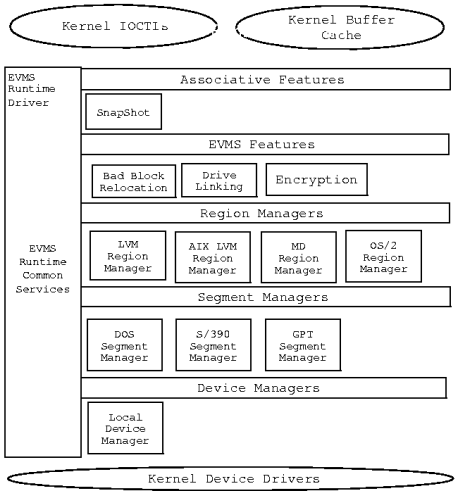

Figure 1: EVMS kernel architecture

the current device manager is sufficient.

The second class of plug-ins is the Segment (or Partition) Managers. Segment Managers are responsible for dividing logical disks into physically contiguous pieces or Segments. A separate Segment Manager is required for each partitioning scheme supported (DOS, S390, Amiga, ...) and only one Segment Manager may be assigned to a disk at a time. It is possible however, to 'stack' Segment Managers, which results in one partitioning scheme being imbedded within a segment created using another partitioning scheme.

The third class of plug-ins is the Region Managers. Region Managers consume disks and/or segments and produce regions which represent logically contiguous storage space. The region layer is the layer in which containers or groups are implemented, making this the place where AIX and Linux LVM plug-ins are found. Also, since this is the last layer before EVMS specific on-disk data structures are introduced, and the first which supports logically contiguous space, it is where most of the compatibility plug-ins are found, including MD. Region Managers, like Segment managers can be stacked, allowing for configurations such as LVM on RAID.

It should be noted that the Device Managers, Segment Managers and Region Managers have no constraints on their size, placement, or format of metadata. Depending on what the plug-in is trying to accomplish, such as compatibility with DOS partitions or Linux LVM, the plug-in can layout metadata in any format desired or required. The advantage of this is that these types of EVMS plug-ins can support any native metadata format. The downside is that with some non-EVMS metadata formats, redundancy and error checking may not be as complete as desired.

The fourth class of plug-ins is Features. This layer contains plug-ins which are designed specifically for EVMS and make use of metadata layouts provided by and enforced by the EVMS common code. Features consume one or more objects created by any of the layers and produce a new feature object. All features share some common metadata, called Feature Headers, on disk. Thus, it is possible to do consistency checking and to even detect if a feature which was configured on disk is for some reason not compiled into the kernel. The Feature Header is where common information like the volume name, volume minor number, feature id, feature depth or count, and versioning information is kept.

Kernel Discovery process

EVMS supports in kernel discovery. This means that each kernel plug-in contains the code required to probe for and recognize the metadata on disk and build volumes without user space intervention. This capability allows an EVMS enabled kernel to recognize and export all volumes that exist in a system at boot time without requiring a RAM disk or any user scripts to be run. This also means that EVMS is not dependent on any user space configuration files that could be accidentally deleted or damaged by file system or other errors. There is some debate as to whether this function belongs in the kernel or in user space. The main reasons for putting it in the kernel is that it make boot setup much easier (no scripts or RAM disks required) and it makes

upgrades easier since the kernel is not dependent on user tools to discover volumes; thus the kernel can be upgraded independently of the user tools.

The kernel discovery process starts with the device manager examining the gendisk list for acceptable devices. The device manager creates a object representing each device to be managed and adds this node to the discover list. This list is returned to the common services which then passes it on to each Segment Manager. Each Segment Manager can examine each node and remove it from the list if it claims the disk. Once a Segment Manager claims a device, it processes this device and creates new objects for each Segment on the device. These new objects are placed on the discover list for further processing. Once each Segment manager has had a chance to claim devices, if any new objects were placed on the list, this process is repeated with the new list. This allows for nested partitioning. This process continues until no new objects are created and then the common services takes the object list (which now consists of disk and segments) and repeats this process with Region Managers.

Once Region discovery is completed, the kernel starts feature discovery. Due to the common Feature Headers for EVMS features this process is much more streamlined and efficient. Each object is examined for the existence of Feature Headers and if one is found, that object is grouped together with other objects with the same volume serial number (found in the Feature Header). Next the Feature Header depth is checked in each of the objects for the volume, and the objects with the deepest or bottom most count are given to the plug-in which will own the objects. This is done based on a feature ID stored in the Feature Header. This process repeats for each level of the feature stack, or until an error is encountered.

EVMS has the concept that only volumes and not intermediate objects are actually exported from the kernel. This helps prevent accidental access to objects in the middle of a volume stack. Additionally, each storage object can be marked as being a volume or not. If a storage object is not marked as a volume via a bit in the Feature Header, then it is not exported for access from the kernel.

After all features have been applied, the common services adds the volume to the global volume list and makes it available through the EVMS block device using the minor number and name stored in the Feature Header. For volumes which do not have EVMS features applied (compatibility volumes), they are assigned their Linux legacy device name (i.e. hda3, group1/lv1) and are assign the next available minor number.

IO Path

The IO path in EVMS is slightly different from many other block devices in Linux due to the plug-in implementation. The entire EVMS subsystem, including all plug-ins, functions as a single block device driver. What this means is that EVMS does not use the driver queue interface to drive IOs from one EVMS layer, or plug-in, to the next. Instead, each EVMS plug-in exports a function table as part of its initialization process. In this function table are read and write entry points. When an IO is received by EVMS on a volume via the EVMS block device, the common services route this request to the appropriate read (or write) entry point of the topmost object in the volume stack. The plug-in owning that object processes the request by modifying (duplicating, splitting, changing offset, etc.) it and calling the entry point of the object(s) to which the request is now destined. This process repeats for each object or layer in the call stack for this volume, and only when the request gets to the Device Manager is it then re-queued to the device queue of the actual disk driver. This is possible due to the EVMS framework, and removes the requirement for each plug-in to consume valuable system resources such as device major numbers and thus allows an infinite number of internal objects such as partitions.

IOCTLs

EVMS supports two types of IOCTLs, global and volume specific. Global IOCTLs are sent to the EVMS block device and are for commands which are not specific to any one volume or plug-in. They are handled entirely by the EVMS common services. These include commands used by the

Engine such as get version, set debug info level, and rediscover. Volume specific IOCTLs are actually targeted at the minor number of the volume in question. These IOCTLs may be processed by the common services in some cases (such as get block size) or the IOCTL may be passed down the volume stack and processed by each plug-in in the stack (such as delete volume). One Global IOCTL which is worth mentioning is the direct plug-in communication IOCTL. This IOCTL allows for an instance of a plug-in in the Engine to communicate with its corresponding plug-in in the kernel. This allows a plug-in writer to code whatever communication is required, although in most cases this support is not needed by the plug-ins.

3 Engine

The EVMS Engine is the core of the user-space administration tools. The Engine is implemented as a shared library. Like the kernel component of EVMS, the Engine implements the same layered plug-in model. The Engine itself provides the common services and framework, while plug-ins, which are also shared objects, provide the real functionality. The Engine has three sets of APIs defined. First, the Engine library provides the front end API set which is used by the User Interfaces. This application interface provides a single well defined set of entry points through which all manipulation of volumes is done, regardless of the type of volume being configured. Second, the Engine defines the Plug-in APIs. This is the set of functions which must be implemented by each plug-in. This standard API set for plug-ins ensures that new plug-in will integrate with the rest of the system. The third API set is the common services provided by the Engine to the Plug-ins to make their job easier.

Changes to a system with EVMS are accomplished by using one of the user interfaces to open and interact with the Engine. When the Engine is opened, it gets a list of devices via an IOCTL to the EVMS kernel component. The Engine then performs a discovery process which is very similar to that found in the kernel. Each device is probed by the plug-ins and a complete in memory representation of entire system is built. As each plug-in discovers devices or objects that it manages, a tree structure or graph is created.

When reading or writing data to or from devices, the IO request is passed to the plug-in appearing below the current plug-in in the volume tree. The IO request is transformed just as it would be in the kernel IO path. This is done to allow for a complete virtualization of new configurations to occur. For example, an LVM container or group can be created using a RAID5 array which has not been committed to disk, nor is running in the kernel. This allows the user to make any number of changes and configure the system exactly as they would like it before writing any changes to the actual on disk metadata or kernel configuration. If the user decides that they do not like the changes, they can simply quit the Engine session without saving and nothing will have been modified. The user may also commit changes at any point in the configuration process if he so desires. Changes to the system are done by writing metadata to disk by calling down the stack of plug-ins until the device manager is reached, The device manager then writes the data to disk by calling a kernel IOCTL. Once the new metadata is committed to disk, the kernel is told to purge deleted or modified volumes from memory and to go rediscover them from changed metadata on disk. During this process the volumes that have been modified are quiesced temporarily by the common services of the EVMS runtime.

Each plug-in inside the Runtime has a corresponding Plug-in inside the EVMS Engine. These Plug-ins provide administrative capabilities for various kinds of logical volumes. For example, the DriveLinking plug-in inside the Runtime is responsible for I/O through DriveLink devices, and the corresponding DriveLinking plug-in inside the Engine handles creation, modification, and maintenance of DriveLink volumes.

Plug-ins in the Engine are typically more complex than their counterparts in the kernel. This is due to the greater number of APIs required for configuration as well as support for the sophisticated user interface. The Engine plug-ins contain all of the parameter validation code for creating and modifying storage objects. The Engine plug-ins must also duplicate the read/write logic found in

the kernel plug-ins in order to support the complete virtulization of volume configuration in the Engine.

In addition to the four classes of plug-ins found in the kernel, there is one additionsl class which is unique to the Engine. This class is File System Interface Modules or FSIMS. FSIMS are a special plug-in class and have their own unique function table. FSIMS are used to allow EVMS to communicate with various File Systems to coordinate changes to volumes. Having an FSIM not only allows actions like mkfs and fsck to be performed directly from the EVMS interface, but it also ensure that actions such as expanding or shrinking a volume are properly coordinated with the File System without the user being required to know the order in which operation must be performed. This capability does not exists anywhere else in Linux.

4 User Interfaces

Because the EVMS Engine provides a programmatic interface instead of a direct user interface, multiple user interfaces can be written or tailored to a particular style or group of tasks. Currently, four different user interfaces have been developed:

1. A general command line which is useful for automating tasks.

2. A graphical user interface (GUI) for easy administration using a GUI desktop.

3. A text-mode, ncurses-based interface.

4. A set of command line utilities for emulating the Linux LVM (logical volume management) command set.

The first three of these interfaces are general purpose and support all existing plug-ins. The fourth, the command line utilities that emulate Linux LVM, is one example of how the Engine application programming interface (API) can be used to create a tailored user interface. As the name implies, these utilities will only interact with the LVM plug-in to manage LVM containers and regions.

All of the user interfaces interact with EVMS through the Engine application APIs. These APIs abstract the specific options and parameters of each Engine plug-in into a well defined interface known as the task interface. A set of well known tasks have been defined (create, expand, shrink, modify, ...) to allow for a negotiation to occur between the user interface and the plug-in. This allows the user interface to write one set of screens or functions for each task, but be able to use this code with any plug-in. It is the plug-in's responsibility to return to the user interface all objects that can be used in a task, and once an object(s) is selected, to return a list of options to the user interface. Each option is described by an option descriptor data structure which allows the user interface to select the proper display method and entry type for the particular option. The descriptor also indicates any restriction or limits on the option such as minimum or maximum values. As each option is set by the user interface, the information is passed to the plug-in for validation. As part of the negotiation or validation, the plug-in can enable or disable other options, or change the values possible for other options.

One of the additional advantages of this approach is the ability for the user interfaces to enforce limits based on actual constraints instead of theoretical ones. For example if there is only 100M of freespace available for a create command, the GUI will not let you enter any value greater that 100M, rather then letting you enter a larger value and fail the command.

5 Future Work

Two main work items will affect the EVMS architecture in the coming year. The first is a generic move capability. New APIs will be added to allow the moving of one storage object to another while the volume is mounted. For example this capability

will allow for the moving of data on a mounted partition to be moved to a LVM region or vise versa. This same technology will be used within plug-ins to provide functionality equivalent to the Linux LVM's pvmove.

The second, and larger work item is cluster support. EVMS is currently working on adding cluster support for configuration of shared storage within a cluster. With this support, plug-ins such as snapshotting and bad block relocation will be able to work on volumes backed by shared storage and access from multiple nodes in a cluster.

6 References

1. The Home Page for the EVMS Project *http://evms.sf.net*
2. EVMS HowTo *http://evms.sf.net/howto*
3. Linux Partition HowTo *http://tldp.org/HOWTO/mini/Partition.html*
4. Software Raid HowTo *http://tldp.org/HOWTO/Software-Raid-HOWTO.html*
5. Linux LVM *http://www.Sistina.com/products_lvm.htm*